Faculty of Mathematical Sciences

University of Twente

University for Technical and Social Sciences

P.O. Box 217
7500 AE Enschede
The Netherlands
Phone: +31-53-4893400
Fax: +31-53-4893114
Email: memo@math.utwente.nl

Memorandum No. 1509

Hierarchical index sets in algebraic modelling languages

J.J. Bisschop, J.B.J. Heerink[1] and
G.H.M. Roelofs[1]

December 1999

ISSN 0169-2690

[1]Paragon Decision Technology, Haarlem, The Netherlands

Hierarchical Index Sets in Algebraic Modeling Languages

J.J. Bisschop*, J.B.J. Heerink**, G.H.M. Roelofs**

Faculty of Mathematical Sciences *
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Paragon Decision Technology B.V. **
P.O. Box 3277
2100 DG Haarlem
The Netherlands

Multi-dimensional algebraic modeling languages make extensive use of simple and compound index sets. In this paper the multi-dimensional modeling paradigm is extended with the concept of a hierarchical index set to support the use of hierarchical data structures. The appropriate reference and indexing mechanisms are introduced, together with mechanisms to support various set operations. Special attention is paid to the Cartesian product of two hierarchical index sets. The modeling of multi-stage programming models is supported through the introduction of a hierarchical indexing mechanism. The extensions proposed in this paper are compared to existing facilities designed to support the modeling of hierarchical structures.

Keywords: Index Sets, Hierarchical Sets, Modeling Languages, Multi-stage Programming.

AMS Subject classification: 68N20, 68N99.

1. Introduction

Index sets play an important role in mathematical programming modeling languages (see e.g. [3], [6] and [1]). In [8], index sets are characterized as finite sets with elements describing model identities with similar characteristics, allowing model fragments to be represented in a 'population independent' way (i.e. independent of instantiating element values). Instead of specifying individual model identifiers (such as variables, parameters

and constraints) as scalar identifiers, index sets permit the specification of groups of related model identifiers. Expressions throughout a model can then be written in terms of these grouped model identities, resulting in an overall concise and clear notation.

The particular choice of index sets used to formulate a model is determined by the modeler, and is a subjective choice. Modeling experience, structure recognition and model maintenance issues all contribute to the final choice. The one example of indexing that should not be used is the model formulation from the solver's point of view. In such a formulation, there is only one index set to reference all model variables and only one index set to reference all model constraints. The actual model formulation then contains just a single symbolic constraint. Such a solver's form is difficult to understand, and instead, a modeler's form using multiple index sets is recommended (see the seminal paper by Fourer [5]).

The elements of index sets in most algebraic modeling languages are atomic, and cannot be sets themselves. In this paper the concept of an hierarchical index set will be introduced to allow the recursive notion of sets of sets. This extension of traditional index sets simplifies the modeling of hierarchical structures, thereby adding to the expressiveness of algebraic modeling languages. Typical examples in which hierarchical structures play a role are organization charts, product assembly flows, social accounting structures, multi-stage scenario representations in stochastic programming, and branch and bound search trees in integer programming. With the exception of LPL [10], modern modeling languages offer little or no support for hierarchical structures. Section 4 will enunciate the differences between available hierarchical language constructs in modeling systems such as AIMMS [1], AMPL [6] and LPL [10].

A formal description of hierarchical index sets is contained in Section 2. In Subsection 2.1 the basic terminology and definitions related to hierarchical index sets are presented. Both the syntax and the semantics associated with these sets are explained, and subsequently illustrated through small examples. Then, a path-based reference mechanism for members is introduced in Subsection 2.2, which forms the basis for explaining the well-known set operations such as union, intersection, etc. applied to hierarchical index sets. Special attention is paid to the Cartesian product involving two hierarchical index sets. In Subsection 2.3 various indexing schemes designed for hierarchical structures are introduced. A special type of indexing to support the modeling of multi-stage programming formulations is proposed in Section 3. Finally, a comparison between existing concepts to support hierarchical structures is provided in Section 4.

2. A Formal Description of Hierarchical Index Sets

2.1. Terminology and Definitions

It always helps to visualize a structure before studying its abstract definition. This observation is also true for hierarchical index sets, which can be represented in the form of a tree. Consider the example hierarchical structure displayed in Figure 1. The corresponding tree has a root node, intermediate nodes and leaf nodes. All nodes have an associated *name*. This name need not be unique, as can be observed from the nodes named b_3. There is the restriction, however, that the sequence of node names along the path from the root to any particular node is unique. This implies that the collection of nodes emanating from a particular node all have unique names, and thus form a set. This requirement is satisfied in Figure 1. The name H associated with the root node identifies the entire hierarchical set. All intermediate and leaf nodes are said to be *members* of this set. In addition to being a member, each such node is also an *element* of its parent set. Thus, each intermediate node is also a set.

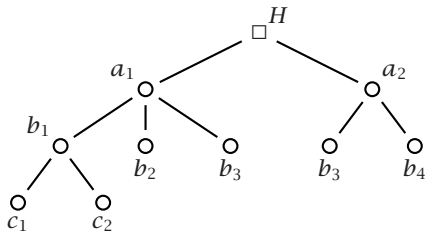
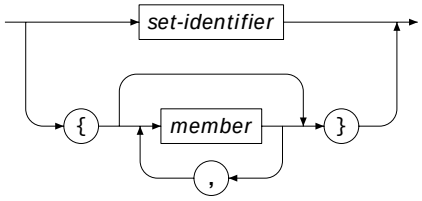


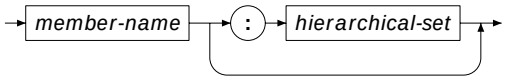
Figure 1. Graphical representation of a hierarchical set H

Based on the observations made in the previous paragraph, the following definitions will be used in the sequel. Let H denote a hierarchical set. Then a *member* of H is an element of H, or recursively, an element of a member of H. A *member name* is either a simple element with one component or a compound element with $n \geq 2$ components. The *member dimension* is the number of components in a member name. The following rules determine the syntax of an hierarchical set.

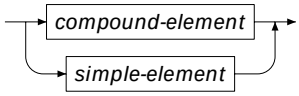
hierarchical-set :



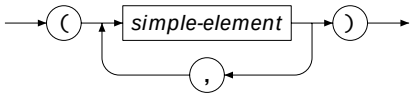
member :



member-name :



compound-element :



The following semantic rules are added to the above syntactic diagrams in order to create workable constructs for practical use inside a modeling language framework. First of all, the number of members in H must be finite. This is a common requirement resulting in finite model instances. Secondly, for every pair of members e_1 and e_2 with parent A, where $A = H$ or A is a member of H, their corresponding member names must not be equal. This requirement guarantees that the corresponding collection of member names is a proper set, and that the sequence of member names from the root to a particular node uniquely identifies each member within H. The third requirement is that all members must have the same member dimension, which is consistent with existing set and indexing notation for multi-dimensional structures.

The following constructs are examples of valid hierarchical index sets, and satisfy both the syntactic and semantic rules described above.

```
H1 := { a, b, c } ;
H2 := { (a1,a2), (b1,b2), (c1,c2) } ;
H3 := { a, t : { b, c } } ;
H4 := { (a1,a2), (t1,t2) : { (b1,b2), (c1,c2) } } ;
H5 := { a, t : { b, c }, u : { b, c } } ;
H6 := { t : H3, u : H5 } ;
```

The following three constructs are examples of invalid hierarchical index sets, because they violate the semantic rules concerning finiteness, element uniqueness and dimension consistency, respectively.

```
I1 := { a, b : I1 }
```

```
I2 := { a, a, c };
I3 := { a, (b1,b2) };
```

2.2. Set operations

As member names are not necessarily unique within a hierarchical index set, they cannot be used as member identification. Instead, each member has a *member ID*, which is defined as the '\'-separated list of all member names from the root node to that particular member. The member ID is also referred to as the *path description* of a member. Consider the following example containing a hierarchical index set H with fifteen members.

```
H := { t : { a, t : { b, c, d } },
       u : { a, s : { b, c }, t : { b, c ,e } } };
```

The corresponding set of full member ID's (path descriptions) is then as follows.

All Path Descriptions				
t	t\t\b	u	u\s\b	u\t\b
t\a	t\t\c	u\a	u\s\c	u\t\c
t\t	t\t\d	u\s	u\t	u\t\e

Note that the member names u, s, d and e are unique within H, and that they also could have served as member ID's. As suggested already in [13], an implementation of member ID's in a modeling language should support abbreviated forms. By replacing a redundant intermediate portion of a path description with .., or eliminating a redundant first portion all together, the member ID $u\backslash t\backslash e$ can also be written as $u\backslash..\backslash e$, $..\backslash e$, or just e. Similarly, the member ID $u\backslash s\backslash c$ can be written as $..\backslash s\backslash c$ or $u\backslash..\backslash c$, but not as c.

There is a correspondence between a hierarchical set H and the set of path descriptions N representing all members of H. Consider the two functions `members`(H) and `hierarchy`(N). The function `members` has a hierarchical index set H as its input, and the set of path descriptions N belonging to all members of H as its output. Similarly, the function `hierarchy` has a set of path descriptions N as its input, and the corresponding hierarchical index set H as output. As a result, $H \equiv$ `hierarchy`(`members`(H)). Note that whenever a set N of path descriptions has an element e_1 that is a left-justified substring of another element e_2 in N, then element e_1 is redundant input of the function `hierarchy`. To describe the two functions in detail is a nontrivial exercise, but for the purpose of this paper it is sufficient to acknowledge their existence. As it turns out, these two functions play a central role in describing the standard set operations on hierarchical index sets.

Let A and B denote two hierarchical index sets. Then the following definitions describe the hierarchical index set operations union ($\cup_{\mathrm{H}}$), intersection ($\cap_{\mathrm{H}}$) and difference ($\setminus_{\mathrm{H}}$) applied to these two sets.

$$A \cup_{\mathrm{H}} B \equiv \texttt{hierarchy}(\texttt{members}(A) \cup \texttt{members}(B))$$
$$A \cap_{\mathrm{H}} B \equiv \texttt{hierarchy}(\texttt{members}(A) \cap \texttt{members}(B))$$
$$A \setminus_{\mathrm{H}} B \equiv \texttt{hierarchy}(\texttt{members}(A) \setminus \texttt{members}(B))$$

It is straightforward to verify that the definitions of ($\cup_{\mathrm{H}}$), ($\cap_{\mathrm{H}}$) and ($\setminus_{\mathrm{H}}$) coincide with their counterparts ($\cup$), ($\cap$) and ($\setminus$) whenever A and B are standard index sets (i.e. the special instance of hierarchical index sets with only atomic members).

Consider the following hierarchical index sets A and B, together with their corresponding sets of member ID's.

```
A := { a, t : { b, c }, d };
B := { a, t : { b, e }, d : { f } };
```

with

```
members(A) = { a, t, t\b, t\c, d };
members(B) = { a, t, t\b, t\e, d, d\f };
```

then the above set operations evaluate to

```
A union B        = hierarchy( { a, t, t\b, t\c, d, t\e, d\f } )
                 = { a, t : { b, c, e }, d : { f } }
A intersection B = hierarchy( { a, t, t\b, d } )
                 = { a, t : { b }, d }
A difference B   = hierarchy( { t\c } )
                 = { t : { c } }
B difference A   = hierarchy( { t\e, d\f } )
                 = { t : { e }, d : { f } }
```

Even though the above set operations for hierarchical index sets are natural extensions of their counterparts for standard index sets, the extension of the Cartesian product between two standard index sets to a Cartesian product between two hierarchical index sets is less obvious. In this paper, both a level-oriented and an atom-oriented Cartesian product between two hierarchical index sets will be developed. The two products have different characteristics, but share the properties that the resulting member names have

the same increased dimension and that the structure of the new hierarchical index set is not affected by the order of the operands.

The level-oriented Cartesian product between two hierarchical sets A and B, each containing more than one level of members, will be denoted with the symbol $\times_{\mathrm{L}}$. The function `atoms` determines the building blocks. The Cartesian product $A \times_{\mathrm{L}} B$ is based on forming all possible tuples with an atom description from A in the first component and an atom description from B in the second component. Each tuple with two path descriptions is then translated into a path description containing several tuples as follows.

$$(t_1 \backslash \ldots \backslash t_m \ , \ u_1 \backslash \ldots \backslash u_n) = \begin{cases} (t_1, u_1) \backslash \ldots \backslash (t_m, u_n) & \text{if } m = n \\ () \quad \text{(empty tuple)} & \text{otherwise} \end{cases}$$

Note that a tuple of two path descriptions only survives when the two path descriptions are of equal length. This requirement makes sure that level-oriented matching of member names always leads to a well-defined path of tuples. Let `levelcombine` denote the function that applies the above process to a collection of tuples with path descriptions as their components. Then the hierarchical Cartesian product $A \times_{\mathrm{L}} B$ can be defined as follows.

$$A \times_{\mathrm{L}} B \equiv \texttt{hierarchy}(\texttt{levelcombine}(\texttt{atoms}(A) \times \texttt{atoms}(B)))$$

Consider the following example as an illustration of the Cartesian product ($\times_{\mathrm{L}}$) between two hierarchical index sets. In this example, the path descriptions of the atomic members in each of the two sets are not of equal length.

```
Transport := { train,
               bikes : { Batavus },
               cars  : { Renault : { R-4, R-6 },
                         Audi    : { A-4, A-8 },
                         Ford }                  };
Colors    := { light : { white, yellow },
               dark  : { black, blue }           };
```

After writing out all atoms of the two sets `Transport` and `Colors`, taking their Cartesian product, and then applying the transformation process deployed by the function `levelcombine`, the resulting Cartesian product `Transport` $\times_{\mathrm{L}}$ `Colors` becomes

```
Transport levelcross Colors  =
  { (train,light), (train,dark),
    (bikes,light) : { (Batavus,white) , (Batavus,yellow) },
    (bikes,dark)  : { (Batavus,black) , (Batavus,blue)   },
    (cars,light)  : { (Renault,white) , (Audi,white) , (Ford,white)  ,
                      (Renault,yellow), (Audi,yellow), (Ford,yellow) },
    (cars,dark)   : { (Renault,black) , (Audi,black) , (Ford,black)  ,
                      (Renault,blue)  , (Audi,blue)  , (Ford,blue)   } }
```

Note that the new hierarchical index set `Transport` $\times_{\mathrm{L}}$ `Colors` has only two levels, and that the third level of the set `Transport` is no longer considered due to the effect of the `levelcombine` function. The natural hierarchy between the individual components among the levels in the new set remains preserved, and the dimension consistency of the new members is guaranteed. If the Cartesian product `Transport` $\times_{\mathrm{L}}$ `Colors` should also include the third-level members of the set `Transport`, it is possible to extend the hierarchy in the set `Colors` to add a third level which repeats the second level. This kind of set extension may seem artificial at first, but is nothing more than expressing in a controlled fashion, how each level of the two originating hierarchies should be combined to form the new (hierarchical) Cartesian product.

A second atom-oriented Cartesian product between two hierarchical sets A and B can also be defined,and will be denoted with the symbol $\times_{\mathrm{A}}$. The function `atoms` again determines the building blocks, and the Cartesian product $A \times_{\mathrm{A}} B$ is also based on forming all possible tuples with an atom description from A in the first component and an atom description from B in the second component. Each tuple with two path descriptions is then translated into a path description containing several tuples as follows.

$$(t_1 \backslash \ldots \backslash t_m \ , \ u_1 \backslash \ldots \backslash u_n) = \begin{cases} (t_1,u_1)\backslash \ldots \backslash (t_m,u_n) & \text{if } m = n \\ (t_1,u_1)\backslash \ldots \backslash (t_m,u_m)\backslash (t_m,u_{m+1})\backslash \ldots \backslash (t_m,u_n) & \text{if } m < n \\ (t_1,u_1)\backslash \ldots \backslash (t_n,u_n)\backslash (t_{n+1},u_n)\backslash \ldots \backslash (t_m,u_n) & \text{if } m > n \end{cases}$$

Note that a tuple of two path descriptions always survives even though the input path descriptions are of unequal length. Let `atomcombine` denote the function that applies the above process to a collection of tuples with path descriptions as their components. Then the hierarchical Cartesian product $A \times_{\mathrm{A}} B$ can be defined as follows.

$$A \times_{\mathrm{A}} B \equiv \texttt{hierarchy}(\texttt{levelcombine}(\texttt{atoms}(A) \times \texttt{atoms}(B)))$$

When applying the atom-oriented Cartesian product to the two hierarchical index sets `Transport` and `Colors`, the following resulting set is obtained.

```
Transport atomcross Colors  =
  { (train,light) : { (train,white)   , (train,yellow)   },
    (train,dark)  : { (train,black)   , (train,blue)     },
    (bikes,light) : { (Batavus,white) , (Batavus,yellow) },
    (bikes,dark)  : { (Batavus,black) , (Batavus,blue)   },
    (cars,light)  : { (Renault,white)  : { (R-4,white)  , (R-6,white)  },
                      (Audi,white)     : { (A-4,white)  , (A-8,white)  },
                      (Ford,white),
                      (Renault,yellow) : { (R-4,yellow) , (R-6,yellow) },
                      (Audi,yellow)    : { (A-4,yellow) , (A-8,yellow) },
                      (Ford,yellow) },
    (cars,dark)   : { (Renault,black)  : { (R-4,black)  , (R-6,black)  },
                      (Audi,black)     : { (A-4,black)  , (A-8,black)  },
                      (Ford,black),
```

```
(Renault,blue)   : { (R-4,blue)   , (R-6,blue)   },
(Audi,blue)      : { (A-4,blue)   , (A-8,blue)   },
(Ford,blue) } }
```

The choice between the two Cartesian products introduced in this paper is application-dependent. Take for instance, an application in which both hierarchical structures employ the same time dimension at each level. Then it is natural to prefer the level-oriented Cartesian product, because mixing information between time levels is without meaning. In the above example, one could argue in favor of either Cartesian product depending on subsequent data computations to be performed.

2.3. Set indexing

Hierarchical index sets can be considered as an extension of standard multi-dimensional index sets, and have a more complex structure. Within this more complex structure there are several subsets of members that are natural candidates for access by a modeler. Examples of special subsets are the set of all members emanating from a particular member, the set of atomic members emanating from a particular member, the set of ancestors of a particular member, etc. The following table provides an overview of functions characterizing specific parts of a hierarchical index set.

function	input	output
`members`	hierarchical-set | member ID	set of member ID's
`elements`	hierarchical-set | member ID	set of member ID's
`atoms`	hierarchical-set | member ID	set of member ID's
`ancestors`	member ID	set of member ID's
`parent`	member ID	member ID
`ancestor`	member ID, number	member ID
`membername`	member ID	member name
`level`	member ID	number
`sublevel`	hierarchical-set | member ID, number	set of member ID's

Using the functions of the above table the following examples illustrate some straightforward referencing of members within a hierarchical index set H using the indices i and j, and the parameter P defined over the members of H. The identifiers on the left of the assignment statements should be self-explanatory.

```
NumberOfElements         :=  count[i in elements(H)];
NumberOfElements         :=  count[i in H];                    ! Is default
NumberOfMembers          :=  count[i in members(H)];
NumberOfScenarios        :=  count[i in atoms(H)];
LargestValueMember       :=  argmax[i in H, P(i)];
MaximumDepth             :=  max[i in atoms(H), level(i)];
```

```
SumOverLeafs(i in H)     :=  sum[j in atoms(i),   P(j)];
SumOverMembers(i in H)   :=  sum[j in members(i), P(j)];
SumOverChildren(i in H)  :=  sum[j in i, P(j)];
SumOverAncestors(i in H) :=  sum[j in ancestors(i), P(j)];
SpecificMemberNameSum    :=  sum[i in members(H) | membername(i)='a', P(i)];
ThirdLevelSum            :=  sum[i in sublevel(H,3), P(i)];
```

Typical operations across hierarchical structures are *aggregation* and *disaggregation*. When detailed information has been collected at the level of leaf nodes, it is natural to view this information at various levels of aggregation inside the hierarchy. The following procedure demonstrates the ease of specifying such an aggregation scheme for summation within a hierarchical index set H using the parameter `Value` defined over the members of H. The other identifiers are self-explanatory.

```
Procedure AGGREGATE
  MaxDepth :=  max[i in atoms(H), level(i)];
  Depth    :=  MaxDepth - 1;
  WHILE (Depth >= 1) DO
    Value(i in sublevel(H,Depth)) :=  sum[j in i, Value(j)];
    Depth                         -=  1;
  ENDWHILE;
```

Similarly, when detailed information has been collected at the top (first) level of an hierarchical index set H, it is natural to distribute this information over the levels underneath. As before, let `Value` denote the identifier with known numeric values initialized at the first level of H. In addition, let F, also defined over the members of H, be the fraction associated with each member to be used to divide `Value` over element members. The following procedure illustrates how a straightforward disaggregation scheme can be constructed.

```
Procedure DISAGGREGATE
  MaxDepth :=  max[i in atoms(H), level(i)];
  Depth    :=  2;
  WHILE (Depth <= MaxDepth) DO
    Value(i in sublevel(H,Depth)) :=  F(i) * Value(parent(i));
    Depth                         +=  1;
  ENDWHILE;
```

3. Hierarchical Indexing and Multi-Stage Models

The indexing examples described in the previous section are essentially examples of member-oriented indexing. In this section, yet another form of indexing is introduced, namely hierarchical indexing, which can be characterized as path-oriented indexing.

Consider a hierarchical set H, and an index s defined over the member set `atoms(H)`. In the context of multi-stage modeling this latter set is sometimes referred to as the set of scenarios. Consider also an ordered indexed set $V(s)$ containing all members along the path from the root to an atomic member s in H in that order. Let m be an index defined over members of H. Then a *hierarchical index* i in H is defined as a 2-dimensional compound index with tuples (m,s), $m \in V(s)$. In addition, the *hierarchical lag operator* $-$ used in the reference to a tuple $i-1$ is defined as the reference to the parent tuple $(\texttt{parent}(m),s)$ of i.

Hierarchical indexing can be applied to multi-stage models. Multi-stage models have been studied for decades, and form a special class of stochastic programming models (see e.g. [11], [12] and [14, Chapter 16]). This paper is not meant to define multi-stage models, but only to touch on a few aspects as they are related to hierarchical sets and indexing. There will be a separate paper exclusively devoted to language concepts espcially developed for the representation of multi-stage programming models.

A multi-stage model can be thought of as a collection of multi-period models with added scenario concatenation constraints linking the various multi-period models. These linking constraints are also referred to as non-anticipativity constraints, and can be derived from the scenario tree, i.e. the hierarchical index set capturing all relevant decision states in a multi-level programming application.

A typical balance equation in a multi-period model, with the index i referring to periods, can be written as follows. The notation in this small example is self-explanatory.

```
balance(i) ..
  stock(i) = stock(i-1) + production(i) - demand(i);
```

By using a hierarchical index i, $i \in H$, with H referring to a hierarchical set representing a scenario tree, the same balance equation can then be written as part of a multi-stage model.

```
balance(i) ..
  stock(i) = stock(i-1) + production(i-1) - demand(i);
```

The only difference is the use of the lag operator for the variable `production`, indicating that this decision must be made at the beginning of each stage (control variable [2, Chapter 17]). The values of the variable `stock`(i) and the parameter `demand`(i) are assumed to be known at the end of each stage (state variable). Note that by transforming a standard time index of a multi-period model into a hierarchical index in support of a multi-stage model, the effect on the representation of multi-period constraints is minimal. Of course, extra work has to be done to transform a multi-period model into an operational multi-stage model, but the description of these extra steps are outside the scope of this paper. This purpose of this section has been to merely introduce the con-

cept of a hierarchical index, and to point at its potential use in multi-stage programming applications.

4. Comparison with Existing Languages

Modern modeling languages offer little or no support to model hierarchical structures. In the 80's, the modeling languages AMPL [6], UIMP [4] and LPL [10] were among the first to introduce elementary support for hierarchical structures. In [13] Kuip compares these three modeling languages, and proposes a unifying set concept to model general hierarchical structures. AIMMS [1] was developed during the 90's, but it too has not implemented concepts proposed by Kuip. In this section the proposals in this paper are compared to the analogous concepts in the available languages AIMMS, AMPL, LPL, and in the work of Kuip. The concepts in LPL and the work of Kuip are the most advanced, and will be referred to throughout the remainder. The elementary concepts available in the other two languages will be discussed first.

Both AIMMS and AMPL support the use of *indexed sets*. An indexed set can be viewed as a set of simple sets, and permits the specification of a two-level hierarchical structure. For instance, a one-level aggregation can be written as

```
a(i) := sum[j in T(i), b(j)];
```

This syntax is a special instance of member-oriented indexing discussed in Subsection 2.3. In addition to indexed sets, both AIMMS and AMPL support the use of indexed element parameters. With such parameters, it is possible to capture a multi-level hierarchical structure by defining for each $i \in I$ the element parameter $\texttt{parent}(i)$ that takes its value from the set I. By knowing for each element its parent element, it is possible to write a clean and compact multi-stage programming formulation as was done in [2, Chapter 17]. Despite the compactness of the resulting formulation, it turned out to be difficult and inefficient to use this same notation to derive (in a recursive manner) the unconditional event probabilities from the known conditional event probabilities. This limitation is entirely due to the lack of appropriate indexing facilities, and would be removed once the concepts in this paper are implemented.

The language LPL allows the specification of *tags* as a medium to reference individual members in a hierarchical set. Every component in a tuple that represents a path in a hierarchy can be equiped with such a tag. The use of tags is restricted in the sense that the tag for all members on the same level must be equal. The use of tags in LPL together with a projection operator provides a facility to identify customized subsets of members in the hierarchical set. Kuip generalizes this notion of tags, and introduces the option to attach *labels* to arbitrary members in a hierarchical set. In this paper these tags are referred to as *member names*, and are mandatory instead of optional. As a result, the

representation of a hierarchical index set in terms of member ID's becomes consistent and regular, and completely supports the subset functions described in Subsection 2.3.

The language LPL uses special operators to identify predefined subsets of members like the set of all elements, the set of all members, the set of all intermediate members and the set of all atoms. Kuip uses the concept of *views* to represent subsets of members. In our proposal, special functions are introduced to quickly select all descendants of a members, all ancestors of a member, all members with the same parent, etc. In addition, the proposal in this paper supports the formulation of multi-stage stochastic models through the use of hierarchical indexing.

The proposal in this paper goes beyond the work of LPL and Kuip in that it allows for hierarchical structures involving compound elements and compound indices. In Kuip, the extension to compound structures is not considered. In LPL, the particular choice to represent a hierarchical structure as a collection of tuples, with components referring to levels, complicates the extension to compound hierarchical structures. Consider, for instance, the Cartesian product of two hierarchical sets. In this paper, such a product extends member names into tuples, providing an increase in dimension for every member name. In LPL, however, the Cartesian product will result in a hierarchical index set in which the second operand set is appended to each leaf node in the first operand set. Member names do not increase in dimension, and as can be seen in the example below, the new structure does not relate the two hierarchies in a practical sense. Consider the Cartesian product of the hierarchical sets `Transport` and `Colors` in Subsection 2.2. In LPL, this Cartesian product leads to the following compound set of tuples (path descriptions).

```
Transport cross Colors =
{ ( train, light, white ),              ( train, light, yellow ),
  ( train, dark, black ),               ( train, dark, blue ),
  ( bikes, Batavus, light, white ),     ( bikes, Batavus, light, yellow ),
  ( bikes, Batavus, dark, black ),      ( bikes, Batavus, dark, blue ),
  ( cars, Renault, R-4, light, white ), ( cars, Renault, R-4, light, yellow ),
  ( cars, Renault, R-4, dark, black ),  ( cars, Renault, R-4, dark, blue ),
  ( cars, Renault, R-6, light, white ), ( cars, Renault, R-6, light, yellow ),
  ( cars, Renault, R-6, dark, black ),  ( cars, Renault, R-6, dark, blue ),
  ( cars, Audi, A-4, light, white ),    ( cars, Audi, A-4, light, yellow ),
  ( cars, Audi, A-4, dark, black ),     ( cars, Audi, A-4, dark, blue ),
  ( cars, Audi, A-8, light, white ),    ( cars, Audi, A-8, light, yellow ),
  ( cars, Audi, A-8, dark, black ),     ( cars, Audi, A-8, dark, blue ),
  ( cars, Ford, light, white),          ( cars, Ford, light, yellow ),
  ( cars, Ford, dark, black ),          ( cars, Ford, dark, blue )}
```

When compared to the Cartesian products listed in Subsection 2.2, it becomes apparent that there are several differences. In the above product set it is difficult to address the hierarchical structure of the originating sets. It is therefore not straightforward to reference members from the second hierarchy (`Colors`) in relation to members of the

first hierarchy (`Transport`). For instance, the selection of all dark cars requires a full search through all tuples involving cars. In both hierarchical sets in Subsection 2.2, this selection corresponds to a single member in the product set. Note that the number of elements (28) in the above set is equal to the number of atoms in the atom-oriented Cartesian product, but their contents is clearly different.

5. Summary

The concepts proposed in this paper are based on the hierarchical set concepts as introduced by Kuip [13]. In this paper the hierarchical index set concept is further extended to support compound hierarchical index sets and the Cartesian product between two hierarchical index sets. In addition, a hierarchical indexing mechanism is introduced to support multi-stage stochastic modeling. It then becomes straightforward to transform a deterministic multi-period model into a multi-stage model.

References

[1] J.J. Bisschop, G.H.M. Roelofs, "AIMMS, The Language Reference", Paragon Decision Technology B.V., The Netherlands, 1999.
[2] J.J. Bisschop "AIMMS, Optimization Modeling", Paragon Decision Technology B.V., The Netherlands, 1999.
[3] A. Brooke, D. Kendrick and A. Meeraus, "GAMS: A User's Guide, release 2.25", Boyd & Fraser/The Scientific Press, Danvers, MA, 1992.
[4] E.F.D. Ellison, G. Mitra, "UIMP - User Interface for Mathematical Programming", ACM Transactions on Mathematical Software, Volume 8, Number 2, 1982.
[5] R. Fourer, "Modeling Languages versus Matrix Generators for Linear Programming", ACM Transactions on Mathematical Software, 9, pp. 143-183, 1983.
[6] R. Fourer, D.M. Gay & B.W. Kernighan, "AMPL, A Modeling Language for Mathematical Programming", Scientific Press, San Francisco, 1993.
[7] R. Fourer, D.M. Gay, "Proposals for Stochastic Programming in the AMPL Modeling Language", Session WE4-G-IN11, International Symposium on Mathematical Programming, Lausanne, August 27, 1997.
[8] A.M. Geoffrion, "Indexing in Modeling Languages for Mathematical Programming", Western Management Science Institute, University of California, Los Angeles, Working Paper No. 371, 1991.
[9] T. Hürlimann, "Hierarchical Index-sets in Modeling Languages", Working Paper, Institute of Informatics, University of Fribourg, Switserland, 1998.
[10] T. Hürlimann, "Mathematical Modeling and Optimization, An Essay for the Design of Computer-based Modeling Tools", Applied Optimization, Vol 31, Kluwer, Dordrecht, The Netherlands, 1999.
[11] G. Infanger, "Planning under Uncertainty", Danvers, Massachusetts, Boyd & Fraser, 1994.
[12] P. Kall, S.W. Wallace, "Stochastic Programming", Chichester, John Wiley & Sons, 1994.
[13] C.A.C. Kuip, "Index Sets in Mathematical Programming Modeling Languages", Ph.d. thesis, University of Twente, The Netherlands, 1992.
[14] H.M. Wagner, "Principles of Operations Research", 2nd Edition, Englewood Cliffs, N.J., Prentice-Hall, 1975.